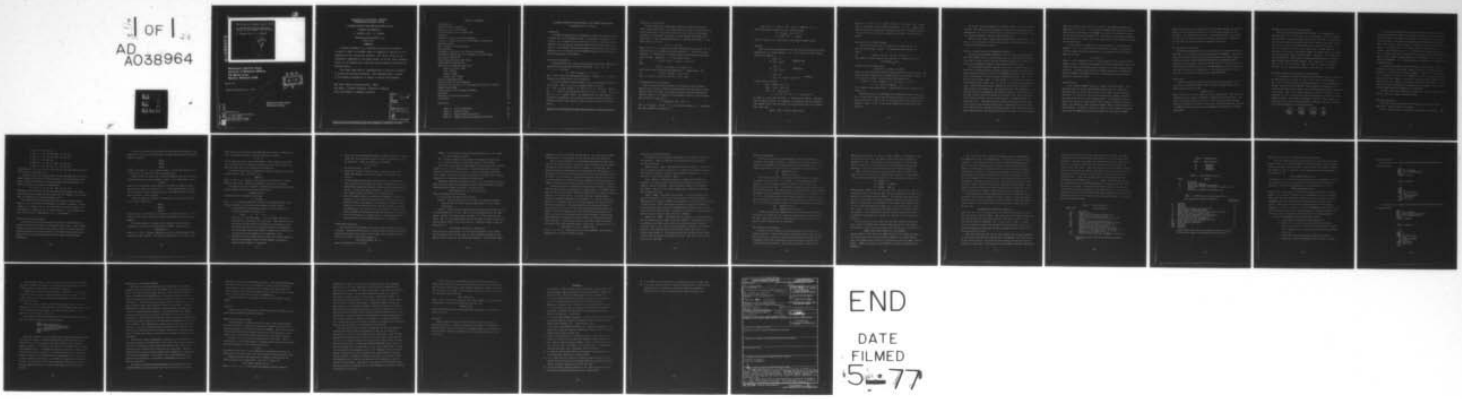
AD-A038 964 WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER F/G 9/2
A FORTRAN-TRIPLEX-PRE-COMPILER BASED ON THE AUGMENT PRE-COMPILE--ETC(U)
MAR 77 K BOEHMER, R T JACKSON DAAG29-75-C-0024
UNCLASSIFIED MRC-TSR-1732 NL

1 OF 1
AD A038964



END
DATE FILMED
5-77

MRC Technical Summary Report #1732

A FORTRAN-TRIPLEX-PRE-COMPILER BASED ON THE AUGMENT PRE-COMPILER

K. Boehmer and R. T. Jackson

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

March 1977

(Received February 22, 1977)

Sponsored by
U. S. Army Research Office
P. O. Box 12211
Research Triangle Park
North Carolina 27709

UNIVERSITY OF WISCONSIN - MADISON
MATHEMATICS RESEARCH CENTER

A FORTRAN-TRIPLEX-PRE-COMPILER BASED ON THE AUGMENT PRE-COMPILER

K. Boehmer and R. T. Jackson

Technical Summary Report # 1732
March 1977

ABSTRACT

Triplex arithmetic is a variation of interval arithmetic in which a "main" or rounded value is computed in addition to the endpoints of the containing interval. The "main" value may be considered, depending on the application, to be the "most probable" result of the computation, with the interval bounds indicating the possible error.

This paper describes an implementation of Triplex arithmetic in single and multiple precision. The implementation is based on the Augment precompiler to obtain an easily used package.

AMS (MOS) Subject Classification: 68A10
Key Words: interval analysis, interval arithmetic
Work Unit Number 8 (Computer Science)

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024.

TABLE OF CONTENTS

A FORTRAN-TRIPLEX-PRE-COMPILER BASED ON THE AUGMENT PRE-COMPILER

K. Boehmer and R. T. Jackson

Introduction

A package has been written to implement a triplex data type and triplex arithmetic for use in FORTRAN programs. The package is available in two versions; one in standard single precision, the other in multiple precision, compatible with the multiple precision package available from MRC (see [2] and [3]). This document describes the capabilities and use of the triplex package as it is to be used with the AUGMENT precompiler (see [4] and [5]), and the maintenance of the package.

Round Off and Intervals

Computation with real numbers nearly always means inexact computations generated by round off and truncation procedures. So if $a,b \in \mathbb{R}$, $* \in \{+,-,\times,/\}$ we find

$$\tilde{c} := a \circledast b \quad \text{instead of} \quad c := a * b ,$$

where $a \circledast b$ means the "computational result". Usually $\tilde{c} \neq c$. Interval Analysis is one way to deal with this problem:

In Interval Analysis the real numbers a,b are replaced by intervals $[\underline{a},\bar{a}]$, $[\underline{b},\bar{b}]$, with $\underline{a} \leq \bar{a}$, $\underline{b} \leq \bar{b}$ properly chosen, and then

(1) $$[\underline{c},\bar{c}] := [\underline{a},\bar{a}] * [\underline{b},\bar{b}] := \{\hat{c} \in \mathbb{R} \mid \hat{c} = \hat{a} * \hat{b},\ \hat{a} \in [\underline{a},\bar{a}],\ \hat{b} \in [\underline{b},\bar{b}]\} .$$

For $* = /$ there is the additional restriction that $0 \notin [\underline{b},\bar{b}]$. It is clear that $c = a * b \in [\underline{a},\bar{a}] * [\underline{b},\bar{b}]$. All computations with real numbers are replaced by computations with intervals with properly chosen boundaries. So one ends up with a final interval which is known to contain the desired result.

Sponsored by the United States Army under Contract No. DAAG29-75-C-0024.

Definition of Triplex Type

If one performs many computational steps to get the final interval $[\underline{f},\bar{f}]$, overly conservative, and thereby sometimes useless, results may arise. At the other hand if many single evaluations with reals have to be done, including the unavoidable errors then it often happens that these errors cancel away, so that the approximate value $\tilde{f}$ and the exact value f very often satisfy a relation

$$|\tilde{f} - f| << (\bar{f} - \underline{f}) .$$

There, even if $\bar{f} - \underline{f}$ might be the only guaranteed error bound for $|\tilde{f} - f|$, one loses a lot of information in totally ignoring $\tilde{f}$. A way out of this situation is a combination of interval analysis and "classical computation", called Triplex. Instead of $a \in \mathbb{R}$ one treats triples

(2) $$T := [\underline{a},\tilde{a},\bar{a}] \quad \text{with} \quad a \in [\underline{a},\bar{a}], \quad \underline{a} \leq \tilde{a} \leq \bar{a} .$$

Here

$$\inf(T) := \underline{a}, \quad \text{main}(T) := \tilde{a}, \quad \sup(T) := \bar{a}$$

are called infimum, main value and supremum of T respectively. Now

(3) $$[\underline{a},\tilde{a},\bar{a}] * [\underline{b},\tilde{b},\bar{b}] := [\underline{c},\tilde{a} * \tilde{b},\bar{c}]$$

with $\underline{c},\bar{c}$ from (1) and by (1) and (2) $\tilde{a} * \tilde{b} \in [\underline{c},\bar{c}]$.

A Second Reason for Triplex Type

There is another essential reason for introducing triplex numbers. There are some iterative numerical methods defined for sequences of triplex numbers, having even better convergence properties than the corresponding classical methods. For example the Triplex-Newton version of the classical Newton method converges [8] if

$$x^* \in [\underline{x}_0,\tilde{x}_0,\bar{x}_0] \quad \text{with} \quad f(x^*) = 0$$

and $0 \notin F'([\underline{x}_0,\bar{x}_0])$, where F' is an interval extension of f' and we get the usual quadratic convergence, if

$$\text{span } F'(X) \leq C \cdot \text{span } X \quad \text{where} \quad \text{span } X = \text{span}[\underline{x},\bar{x}] = \bar{x} - \underline{x} .$$

The corresponding condition for the classical method

$$x^* \in [\underline{x}_0,\bar{x}_0] \quad \text{with} \quad f(x^*) = 0$$
$$0 \notin f'(x) \quad \text{for} \quad x \in [\underline{x}_0,\bar{x}_0]$$

does not guarantee the convergence of the classical Newton method.

Rounding

Here we are mainly interested in the case in which we deal with numbers representable on a fixed computer. Let M be the set of these numbers. Then we call mappings

$$r_d : \begin{cases} R \mapsto M \\ a \mapsto r_d(a) \end{cases} \qquad \text{rounding down}$$

$$r_u : \begin{cases} R \to M \\ a \mapsto r_u(a) \end{cases} \qquad \text{rounding up}$$

$$r : \begin{cases} R \to M \\ a \mapsto \tilde{a} \in [r_d(a),\ r_u(a)] \cap M \end{cases} \qquad \text{rounding .}$$

In some cases one defines

$$r_d(a) := \max\{x \in M \mid x \leq a\}$$
$$r_u(a) := \min\{x \in M \mid x \geq a\} .$$
$$r(a) = x \in \{r_d(a), r_u(a)\}$$
$$|x - a| \leq |y - a| \quad \text{for} \quad x \neq y \in \{r_d(a), r_u(a)\} .$$

r_d and r_u are called controlled, r uncontrolled rounding. Implementation of Interval Analysis on a computer, including the goal of guaranteed error bounds, means that the result of every operation has to be rounded in such a way, that

$$[\underline{a},\bar{a}] * [\underline{b},\bar{b}] = [\underline{c},\bar{c}] \subseteq [\bar{r}_d(\underline{c}), r_u(\bar{c})] .$$

Then for $\tilde{a} \in [\underline{a},\bar{a}]$ and $\tilde{b} \in [\underline{b},\bar{b}]$ one has $\tilde{a} * \tilde{b} \in [\underline{c},\bar{c}]$. Now r, r_d, r_u have to be defined in such a way as to guarantee $r(\tilde{a} * \tilde{b}) \in [\bar{r}_d(\underline{c}), r_u(\bar{c})]$. Thus the results of computation in Triplex consist of a reasonable estimate to the actual result as well as accurate error bounds - the best of both worlds.

Different Forms of Triplex Numbers

There are different ways to characterize triplex numbers. We have mentioned already $[\underline{a},\tilde{a},\bar{a}]$. An equivalent way would be the triple

(4) $$\langle \tilde{a},\bar{e},\underline{e}\rangle \quad \text{with} \quad \bar{e} := \bar{a} - \tilde{a},\ \underline{e} := \underline{a} - \tilde{a},$$

for example a triplex number of this type for π could have the form

$$\langle 3.1415926, 1_{10^{-7}}, 0\rangle$$

compared with the original type

$$[3.1415926,\ 3.1415926,\ 3.1415927].$$

That means the second type needs less storage space than the first one. Two other important arguments are the computational and programming expense: It is not hard to see that, using Wilkinson's estimations [11]

$$|a \circledast b - a * b| \leq \varepsilon |a \circledast b|,\ b \neq 0 \quad \text{for} \quad a/b,\ * \in \{+,-,\times,/\},$$

$$\varepsilon > 0, \quad \text{small for example} \quad \varepsilon = 2^{-t}$$

for a computer using numbers with t binary digits and an accumulator of length $2t$.

Computations with (4) need hardly more computational effort than with (2). If one uses a computer with different precisions, (4) certainly needs less time than (2). Then $\underline{a},\tilde{a},\bar{a}$ in (2) have to be in the same precision whereas $\tilde{a}$ (resp. $\bar{e}$, $\underline{e}$) in (4) may be represented in high (resp. low) precision.

If one has to realize triplex on a computer today, it has to be done by software. The computer routines using (4) grow more and more complicated compared with (2). So it is wise to use (2) instead of (4) as long as there is no special hardware to use for triplex computations.

There are many cases in which the intervals blow up very quickly. So when it is very important to have available additional guard digits available, we need multiple-precision-triplex.

Triplex, an Application of the Augment Precompiler

We have used the AUGMENT precompiler [3,4] to define the nonstandard data type TRIPLEX and to provide the user with the corresponding arithmetic. To do this, we have used the interval arithmetic package [9], already available with the AUGMENT precompiler, and have added the main value.

In the multiple precision version, each value of a triplex variable is stored as a multiple precision variable using the multiple precision arithmetic package available for use with the AUGMENT precompiler. This package stores real numbers in a floating point representation using $33 * n - 2$ bits for the mantissa, where n is an integer (>1) that may be changed by reassembling the package, and 33 bits (in all cases) for the characteristic. Thus the precision may be changed over a wide range, although it remains fixed in any single run. The package usually used at MRC has $n = 3$, yielding 97 bits or approximately 29 digits of accuracy. A description of the basic multiple precision package is contained in [2], and a description of the revised interface for use of this package in FORTRAN programs is contained in [3]. The single precision triplex package stores each value of a triplex in the standard floating point representation, but uses many of the basic routines from MRC's interval arithmetic and interval I/O packages, especially the "best possible arithmetic" routines, for its operations.

These are described and listed in [1] and [9]. The AUGMENT precompiler is described in [4] and [5]. It is highly recommended that the user be familiar with the user documentation for AUGMENT [4], and with the revised interface for the multiple precision package [3]. The latter is especially recommended as the user may find useful some of the capabilities of the "multiple" package not described here.

In the language of AUGMENT, the triplex package is a "safe subroutine package". That is, computations are carried out via calls to subroutines with the results returned as parameters in the subroutine call, and statements of the form

$$A = A \text{ op } A ,$$

where A is a variable and "op" is a binary operator, will always yield the desired results. However the programmer need not be concerned with this fact (except of course for realizing that statements of the type indicated work correctly) - programs may be written as though triplex were a standard data type in FORTRAN, and AUGMENT will parse all expressions and generate the appropriate subroutine calls.

Reserved Words

As in standard FORTRAN, there are a number of reserved words. In the multiple precision version the programmer should treat all identifiers beginning with the letters "MPI" or the letters "TPL" as reserved words. The former are identifiers for the multiple precision package, the latter are identifiers for the triplex package itself. In the single precision version all identifiers beginning with "TPS" should be considered reserved words. Two subroutine names, CVTIN and CVTOUT, common to both versions, are also reserved. In addition, there are a number of new "standard" functions recognized by AUGMENT when processing programs for the triplex package.

Those functions which are properly part of the triplex package are described below. All the "standard" functions of the "multiple" package are also available. (For documentation see [3].) The programmer may, if he wishes, use the names of these functions as variable or array names - however he is then unable to reference them as "standard" functions.

The Precompiler Descriptions

The AUGMENT precompiler requires a description of each nonstandard data type that it is to process (see the documentation for AUGMENT). The descriptions for multiple precision and multiple precision triplex may be supplied by ADD'ing library elements (DESCRIPTION/MULTIPLE and DESCRIPTION/TPL) supplied with the package. Note that the description of the multiple precision package must precede that of the TRIPLEX package, since the latter uses the former for many of its computations. The description deck for single precision is available as library element DESCRIPTION/TPS.

Declarations

Triplex variables and arrays are declared to the precompiler in the same manner as MULTIPLE, COMPLEX, REAL, or other variables and arrays. A type declaration has the form:

TRIPLEX ⟨list⟩

where ⟨list⟩ is a list of identifiers to be typed triplex. The type name "TRIPLEX" may also appear in FUNCTION or IMPLICIT type declarations. Note that in the program itself, one need not indicate which triplex package (single or multiple precision) is to be used. The description deck supplied to AUGMENT indicates the precision and AUGMENT will translate the program accordingly.

Arithmetic Operators for Triplex Variables

The usual arithmetic operators for addition (+), subtraction (-), multiplication (*), and division (/), as defined in (2) and (3) above, are available for operations between two triplex variables. For each operation the upper and lower values of the triplex (known as the "sup" and the "inf" of the triplex) are handled exactly as in interval arithmetic; i.e. the sup is rounded up and the inf is rounded down. Normal rounding, i.e. rounding to the nearest representable number, is always used for the middle value (known as the "main") of the triplex. Unary plus (+) and minus (-) are also available. The package also provides exponentiation to integer powers (only) using the usual operator (**), where, if T = (a1,a2,a3), then T ** m = (c1,a2 ** m,c3), with c1 and c3 chosen so that the interval [c1,c3] contains all values x ** m with x in [a1,a3].

Arithmetic Operations for Variables of Other Types

It is important to be able to operate with different types of variables. But what should be the result of, say, T op D with a triplex number T and a real D? The main goal in applying Triplex is to get guaranteed error bounds. Integers are exact numbers, triplex numbers are approximations including guaranteed error bounds, reals are approximations without guaranteed error bounds. Thus A op B must be defined as integer, triplex or real according to A op B being exact, having guaranteed error bounds or having neither property. So we have the following table for A op B where op represents any of the four basic arithmetic operators, and "Real" represents real, double precision, or multiple precision.

A\B	Integer	Triplex	Real
Integer	Integer	Triplex	Real
Triplex	Triplex	Triplex	Real
Real	Real	Real	Real

For operations between triplex and real, double precision, or multiple precision operands (or between real, double precision, or multiple precision and triplex - the order is irrelevant), the operation is carried out in the precision of the nontriplex operand, between that operand and the main of the triplex. Thus the result is always in the type of the nontriplex operand. For example, if D is a double precision variable and T a triplex, then the operation T * D is carried out by converting the main of T to double precision and performing the multiplication in double precision, with the result in double precision. Equivalently, T * D is the same as CTD(MAIN(T)) * D. (See below and [3], respectively, for definitions of MAIN and CTD.)

For operations between integers and triplex quantities the integer is replaced by a triplex having that integer for each of its values, and the operation is carried out in triplex arithmetic. Thus, if I is integer, I * T represents CIT(I) * T or COMPOS(I,I,I) * T. (See below for definitions of CIT and COMPOS.) Note that if single precision triplex is being used then not all integers are expressible exactly as real numbers. In this case the integer is rounded to the appropriate real number for each value of the triplex before the computation is performed.

To obtain the triplex quotient of integers A and B, we may use the function RAT(A,B) which is described below.

Relational Operators

The usual relational operators are available for use with triplex quantities. They are defined as follows:

Let A be the triplex (a1,a2,a3), B the triplex (b1,b2,b3). Then:

A .LT. B <=> (a3 .LT. b1)

A .LE. B <=> (a1 .LE. b1) .AND. (a3 .LE. b3)

A .GE. B <=> (a1 .GE. b1) .AND. (a3 .GE. b3)

A .GT. B <=> (a1 .GT. b3)

A .EQ. B <=> (a1 .EQ. b1) .AND. (a3 .EQ. b3)

A .NE. B <=> (a1 .NE. b1) .OR. (a3 .NE. b3) .

(The operators .LT., .LE., etc. on the right are the usual Fortran operators for Less Than, Less Equal, etc.)

Perhaps it should be noted here that if the programmer desires, he may define additional relational or other operators, or he may redefine those above. Other definitions are certainly possible and perhaps useful in certain contexts; for example, we might define

A .LT. B <=> (a1 .LT. b1) .AND. (a3 .LT. b3) .

This may be done easily by writing the appropriate subroutines to effect the operations and changing the description deck for AUGMENT. For a discussion, see "Modifying the Triplex Package", below.

No relational operations between different types of data have been defined. It is very simple, and adequate, to compare an integer I and a triplex T by CIT(I) .LT. T, but it should be difficult, because it is not adequate, to compare a real R and a triplex T. If one wants to do that, one must write COMPOS(R,R,R) .LT. T, for example.

Conversions Between Data Types

In dealing with conversion between data types one must consider the amount of information available in the different data types. Exact integers contain more information than triplex numbers, which in turn contain more information than real, double precision, or multiple precision quantities. Conversions are only defined from data types to types containing less information.

Conversion functions are available for converting from triplex to real, double precision, and (in the multiple precision version) multiple precision. They are called by

```
CTR(T)
CTD(T)
CTM(T) ,
```

respectively, where T is a triplex variable. In each case the main value of the triplex is converted to the appropriate type.

There is only one conversion to triplex available, that for the conversion from integer to triplex alluded to above. It is called explicitly by

```
CIT(K) ,
```

where K is an integer, and its result is a triplex all three of whose values are set equal to the value of K. In the single precision version not all integers can be represented exactly, but proper rounding is always used to produce a correct triplex.

Three functions are supplied for extracting any of the three values of a triplex. These are:

```
INF(T)
MAIN(T)
SUP(T) ,
```

which extract, respectively, the inf, main, or sup of the triplex T. The result in each case is of the same type as the precision of the triplex package.

The three values of a triplex may be specified, or the triplex "composed", by use of the triplex function COMPOS. The call is via

```
COMPOS(A,B,C) ,
```

where A, B, and C become, respectively, the inf, main, and sup of the resulting triplex quantity. The only restrictions on A, B, and C are

that they must all three be of the same type (real, integer, double precision, or multiple precision), and that they must be ordered:

$$A \leq B \leq C \ .$$

If this ordering does not hold, the triplex is still formed as specified, but in addition an error condition is detected and the action taken is as described below under "Error Conditions and Handling".

A triplex quantity may be formed from a rational number by use of the triplex function RAT. The call is via

RAT(A,B) ,

where A and B are integers. The result of the function is a triplex with the ratio A/B, appropriately rounded, for each of its values.

A triplex may also be specified by a Hollerith string using the triplex function ASSIGN, which may be called explicitly via

ASSIGN('string') ,

or implicitly via a statement of the form

T = 'string' ,

where T is a triplex variable and 'string' is a Hollerith string. The following comments are relevant to the operation of this routine:

1. Strings should be expressed in either of the usual forms for Hollerith constants or arrays, i.e.

   nHccc . . . c or 'ccc . . . c'

   where the c's are characters, n is an integer, and exactly n characters are specified in the first case. (The subroutine doing the conversion also expects the string to be followed by a word containing all 1-bits, but so long as the string is expressed in one of the above formats the Fortran V compiler will generate such a word when the string appears in the subroutine call.)
2. The Hollerith string should contain either one or three numbers in any Fortran-acceptable format for real numbers, separated by commas if three numbers are specified.

3. Any or all of the fields may be empty, in which case zero is assumed, except that the string must contain at least one character (a blank is sufficient). Thus, for example, the statement

```
T = '-1.0,,1.0'
```

builds the triplex (-1,0,1).

4. Except when an entire field is blank, as described above, ALL BLANKS ARE IGNORED, not treated as zeroes as in Fortran input routines.
5. If three numbers are specified, then the first is rounded down to obtain the inf of the triplex, the second is rounded to the nearest representable number for the main, and the third is rounded up for the sup. If only one number is specified, the action is as though that number had been specified in all three positions.
6. Since the ASSIGN function is described to AUGMENT as a conversion function, it will also be called to convert a Hollerith string to TRIPLEX if that string appears as an operand to a binary operator defined for triplex operands and if the other operator is already of TRIPLEX type. See [4], pp. 50-51, for details of the algorithm used by AUGMENT to determine when such conversions are to be invoked.
7. Errors are possible, e.g. bounds errors, illegal characters, etc. See below (Error Conditions and Handling) for a discussion.

INPUT/OUTPUT Subroutines

Two subroutines are provided to facilitate input and output of triplex variables. Subroutine CVTIN converts an input string of Hollerith (FIELDATA) characters representing the value of the triplex into a machine representation of the triplex. The calling sequence is

```
CALL CVTIN (STRING, NC, T) ,
```

where the arguments are as follows:

STRING - an array which contains the Hollerith string in A1 format, i.e. one character per word.

NC - an integer indicating the number of characters in the string

T - a triplex variable into which the result is to be placed

All of the remarks above for the ASSIGN function apply as well to CVTIN except the first - here STRING must be a Fortran array (and no word of 1-bits is necessary after the string). In fact, the function of ASSIGN is to "unpack" the Hollerith string it receives as its input into a Fortran array and to call subroutine CVTIN for the conversion.

Subroutine CVTOUT has the opposite effect - i.e. it converts a triplex from its binary (machine) representation into a Hollerith string (or rather three Hollerith strings) for output. The call is via:

CALL CVTOUT (T, E1, L1, E2, L2, E3, L3) ,

where the arguments are as follows:

T - the triplex variable to be converted

E1,E2,E3 - three arrays which are to receive the decimal representations for the inf, main, and sup, respectively, in A1 format. (See below.)

L1,L2,L3 - three integers which specify the lengths of the three strings

It is permissable for E1, E2, and E3 to be actually parts of the same array. Suppose, for example, that a triplex T is to be converted with 10,15, and 20 characters desired for the inf, main, and sup, respectively, and the output string is to be the array E, which has length at least 45. Then the call

CALL CVTOUT (T,E,10,E(11),15,E(26),20)

will put the inf in positions 1-10, the main in positions 11-25, and the sup in positions 26-45 of the array E, which can then be output in the desired format using nA1 Hollerith specifications. Also note that proper

rounding is used for each value of the triplex: i.e., the inf is rounded down; the main, to the nearest; and the sup, up during the conversion.

The only error possible is not providing enough space for the output string. The single precision conversion routine requires for each number five places plus the number of digits to appear in the fraction. The multiple precision routine uses only as much space for the exponent as is necessary (from zero to twelve columns) plus two for sign and decimal point in addition to the fraction. The multiple precision routine also detects an error condition if too many digits are requested (only about twice the number of significant digits are available), but still returns its maximum.

If more flexibility is required for input/output, the basic conversion routines are available for conversion of single numbers. In multiple precision, use the routines MPICTB, MPITDF, and MPITDV directly from the multiple package - see [3] for details. In single precision, routines almost identical to BPACVI and BPACVO of the interval I/O package are provided. These are not considered part of the triplex package, but are supplied for the user's convenience. The only differences from the descriptions of these routines in [1] are that the names have been changed to TPSCVI and TPSCVO so as not to introduce any new reserved words not beginning with "TPS", and that the calling sequences have been expanded to include both OPTION and FAULT as parameters, so that reference to common block BPAIND is not necessary. Note that this implies that TPSCVI treats all blanks as zeroes, unlike CVTIN and ASSIGN which ignore them. The new call is via:

CALL TPSCVx (E, IW, X, OPTION, FAULT) .

(Here x = I or O.) Definitions for all five parameters are exactly as described in [1] for the original routines.

Additional "Standard" Functions

In addition to the functions listed above, four "library" functions are available. (More are available for multiple precision quantities - see [3] for details.)

Absolute value (ABS). The absolute value of a triplex T has as its main the absolute value of the main of T, and for its containing interval the interval of absolute values of the containing interval of T. It is obtained via the triplex function ABS(T).

Sign (TSIGN). We define the sign of a triplex to be positive or negative if all three values of the triplex are respectively positive or negative, and zero otherwise. Then the integer function TSIGN assumes the values -1,0,+1 if its triplex argument has, respectively, negative, zero, or positive sign.

Square (TSQR). The square of a triplex T is not necessarily the same as the product T * T, as is also the case in interval arithmetic. The square of an interval consists of the squares of all real values contained in that interval, (and is thus always nonnegative), while the product of an interval with itself consists of products of all possible pairs of real values chosen from the interval. The square of a triplex T may be obtained from the triplex function TSQR by writing TSQR(T).

Square root (SQRT). The square root of a triplex T may be obtained via the triplex function SQRT by writing SQRT(T). If the triplex contains any negative values the result is set to the largest possible interval, with main equal to zero, and an error condition is detected - see below for handling. The algorithm used yields the exact square root when it is machine representable, or the upper round for the sup and the lower round for both the inf and main values when it is not. See [12] for details and proof of the algorithm.

Storage Considerations

In the single precision triplex package, each triplex variable is stored in a real array of length 3. Declarations of triplex variables are translated by AUGMENT into the appropriate real array declarations. For example, declaration (a) below will be transformed into declaration (b):

```
(a)     TRIPLEX S,T(3,4)
(b)     REAL S(3),T(3,3,4)
```

In the multiple precision version each triplex variable is stored in an integer array of length 3 * n, where n is the number of words required to store a multiple precision variable in the version of the multiple package to be used. (See [3] for a discussion of multiple precision storage requirements.) Thus declarations of triplex variables are translated by AUGMENT into the necessary integer array declarations. For example if n = 4 (i.e. each multiple precision variable requires four words of storage), and the declaration (a) below is input to the precompiler, then declaration (b) is the translated statement which will appear in the output from AUGMENT:

```
(a)     TRIPLEX S,T(3,4)
(b)     INTEGER S(12),T(12,3,4)
```

It should be noted that AUGMENT does not process DATA or EQUIVALENCE statements, but rather they are copied into the output file exactly as they are read. Hence the use of such statements should take into account the manner in which declarations are translated.

Error Conditions and Handling

A number of error conditions are possible during the processing of triplex operations. These conditions are monitored whenever they may occur, and if one arises a flag is set and the error-handling subroutine (TPSERR or TPLERR in the single and multiple precision versions, respectively), is called for proper handling. The action to be taken in each case is

determined by the value in a table in common (MONTOR) corresponding to the fault which has occurred. The values in this table may be accessed or changed by the user. In addition, the fault indicator, the name of the routine in which the error occurred, and three parameters (usually the parameters in the call to that routine) are made available internally.

Table 1 below contains the list of possible values for TFAULT, the fault indicator, and the error conditions they represent. Values between 1 and 63 represent bounds faults and are to be interpreted as follows: Put

```
F1 = TFAULT / 16
F2 = TFAULT /  4 (MOD 4)
F3 = TFAULT    (MOD 4)
```

(Equivalently, TFAULT = 4 * (4 * F1 + F2) + F3.) Then F1, F2, and F3 represent the fault for the inf, main, and sup, respectively, of the triplex, according to the faults shown in Table 2. (An infinity fault occurs when the size of the result of a single computation exceeds the capacity of the machine representation for the number and the largest representable number is not a valid approximation or bound, e.g. in the calculation of the sup of a triplex. An overflow fault occurs when the size of a result exceeds the capacity of the representation but the largest representable number is an acceptable approximation or bound, e.g. with the inf of an all-positive triplex.)

Access internally to the fault flag and other information about the fault, and to the MONTOR table, which dictates the action to be taken on encountering a fault, is through common block TFAULT, which is declared as:

```
COMMON/TFAULT/TFAULT,NAME,MONTOR(16),PARAMS(3)
```

Variable TFAULT is the fault flag, NAME contains (in A6 format) the name of the routine where the fault was detected, MONTOR is the aforementioned table, and PARAMS usually contains the parameters to routine "NAME" (details below). PARAMS is declared as type TRIPLEX, the other variables all as INTEGERS.

The action to be taken in the error-handling routine is dictated by the appropriate value of the MONTOR table. The possible actions are given in Table 3. The "walk-back" and the "error-counter" mentioned here are part of the error processing utility routines on the Univac 1110 at the University of Wisconsin. Details are given in [6] and [10], but briefly, they operate as follows: A walk-back is a sequence of one-line messages which traces the sequence of calls made back to the main program, and can be a handy debugging aid. The utility routines maintain a count of errors encountered in Fortran library functions - this is the error counter. When the error counter reaches the error limit, processing is terminated. The value of the error counter may be obtained and the error limit may be reset during execution of a program. (The error limit is initialized at 1.)

The correspondence between MONTOR values and error conditions, along with the initial values of MONTOR, are given in Table 4. Note that except for bounds errors these conditions correspond exactly to the error conditions given in Table 1. For bounds errors, the action taken is that indicated by the value

MAX(MONTOR(F1),MONTOR(F2),MONTOR(F3)) ,

where F1, F2, and F3 are as before, the faults for the inf, main, and sup, respectively, of the triplex. Thus if the main of a triplex computation has an underflow fault, the sup an infinity fault, with no fault for the inf, and the values of MONTOR(I), for I = 2,3, are 3 and 2, respectively (the default values), then by Table 3 the error-handler will print an error message and a walk-back and will increment the error counter, the action given by the value 3.

The values ordinarily put into the array PARAMS are the three arguments to the routine where the error was detected. For example, if a fault occurs in the division routine, the values are the dividend, the divisor, and the quotient, in that order. The exceptions are the COMPOS functions, the square

and square root routines, and the input and output routines. The COMPOS functions have the triplex formed by the routine in all three parameter locations. The square and square root routines have only one input parameter, so it appears in both the first two places, with, as usual, the result in the third place. Input errors leave the first parameter undefined, have the triplex that was formed in the third position, and leave the total length of the input expression as an integer in the first word of the second parameter. Note that this may not be consistent with the type declaration for the PARAMS array. If this number is desired internally, it is recommended that an EQUIVALENCE expression be used to access this word directly as an integer. Output errors leave the triplex to be converted in the third parameter, the number of the output string (1, 2, or 3) which caused the error in the first word of the first parameter, and the specified length of that field in the first word of the second parameter. Once again note the nonstandard usage.

Table 1. Fault Conditions

Fault Flag	Fault Condition
0	No Faults
1-63	Bounds Faults
64	Division by a triplex containing zero
65	Badly ordered triplex specified
66	Square root of a triplex containing negative values
67	Insufficient space in output string
68*	Too many digits requested (maximum supplied)
69	Wrong number of fields in input string
70	Illegal character in input string
71	Input string too long to ASSIGN or CVTIN (limits: 80 for sing. prec., 240 for mult. prec.- mult. prec. limit applies only to ASSIGN)
72*	Format error in input string
73*	Input substring too long - truncated string used

* These errors occur only in the multiple precision version.

Table 2. Bounds Faults

Value	Fault Condition
0	No Faults
1	Overflow
2	Infinity
3	Underflow

Table 3. Fault Handling Actions

Value	Action
0	No action
1	Print error message
2	Print error message and walkback
3	Print error message and walkback, increment error counter
4	Print error message and walkback, stop

Table 4. MONTOR Values and Associated Faults

I	Fault Condition affected by MONTOR(I)	Default value of MONTOR(I)
1	Overflow	3
2	Infinity	3
3	Underflow	2
4	Division by a triplex containing zero	3
5	Badly ordered triplex specified	3
6	Square root of a triplex containing negative values	3
7	Insufficient space in output string	1
8*	Too many digits requested for output string	1
9	Wrong number of fields in input string	3
10	Illegal character in input string	3
11	Input string too long to ASSIGN or CVTIN	3
12*	Format error in input string	3
13*	Input substring too long	1
14**	Overflow	3
15**	Infinity	3
16**	Underflow	1

* These errors occur only in the multiple precision version.

** These errors apply to conversions from input strings.

Changing the Precision of the Multiple Precision Version

All the remarks on precision and changing the precision in [3] apply without change to the multiple precision triplex package, since the multiple precision package is used by triplex. When changing the precision, however, one additional step is necessary. After generating the new multiple precision "machine" via

@mfile.NEW-MACHINE,option n

as described in [3], the entire triplex package must be recompiled in order to accommodate the change in storage requirements of multiple precision quantities. This can be accomplished via the following control card:

@tfile.TPLGENERATE,option

where "tfile" is the name of a program file containing the entire triplex package. If no option is present on the TPLGENERATE card, all translated source and individual relocatable elements are lost, with only the element TPLPACKAGE, the collection of all relocatable elements, being saved in "tfile". The option "S" causes all translated source and relocatable elements to be saved in "tfile". Note that TPLGENERATE requires that a copy of the description deck for the multiple package, consistent with the version to be used, be contained in the same file as the triplex package.

A call on TPLGENERATE causes the following actions:

1. Call AUGMENT to translate all the subroutines.
2. Compile all the subroutines, depositing all source and relocatable elements in "tfile", if the option "S" is used, or in a temporary file, (which will be automatically assigned to the run).
3. Collect the compiled subroutines together to form the relocatable element TPLPACKAGE, which is stored in "tfile".

Sample Runstreams

The following is a sample runstream for using the single precision triplex package:

```
@RUN  . . .
@XQT  afile.AUGMENT
@ADD  tfile.DESCRIPTION/TPS
*BEGIN
:FOR,IS  element1
  .
  .
  .
:FOR,IS  element2
  .
  .
  .
  etc.
*END
@ADD  20.
@MAP,IS  ,abs-element
  IN  TPF$.
  IN  tfile.TPSPACKAGE
@XQT  abs-element
  data (if any)
@FIN
```

The following is a sample runstream for using the multiple precision triplex package:

```
@RUN  . . .
@XQT  afile.AUGMENT
@ADD  mfile.DESCRIPTION/MULTIPLE
@ADD  tfile.DESCRIPTION/TPL
*BEGIN
:FOR,IS  element1
  .
  .
  .
:FOR,IS  element2
  .
  .
  .
  etc.
*END
@ADD  20.
@MAP,IS  ,abs-element
  IN  TPF$.
  IN  tfile.TPLPACKAGE
  LIB  mfile.
@XQT  abs-element
  data (if any)
@FIN
```

The filenames "afile", "mfile", and "tfile" are names of files which contain the AUGMENT precompiler, the multiple precision package, and the triplex package, respectively, and need not be distinct.

These runstreams assume that "element1", "element2", etc. are placed in TPF$ and that no other routines from other files are needed. If this is not the case the control statements following the @MAP card should be altered accordingly.

Note the colon in Column 1 of the Fortran control cards. This is necessary in order for these cards to be read by AUGMENT without interference from the Exec 8 operating system. The colon will be changed to a 7-8 punch ("@") before output to File 20.

The following runstream will generate a new multiple precision package and a new triplex package with a new precision:

```
@RUN  . . .
@mfile.NEW-MACHINE,option n
@COPY,S mfile.DESCRIPTION/MULTIPLE,;
        tfile.DESCRIPTION/MULTIPLE
@tfile.TPLGENERATE,option
@PREP  mfile.
@FIN
```

Here again "mfile" is a file containing the multiple precision package (including the symbolic elements which may have to be loaded from tape) and "tfile" is a file containing the triplex package (also containing symbolic elements). If these are the same file, the @PREP card must follow the TPLGENERATE card. (The @PREP card causes an entry point table to be generated for the file, which will be destroyed whenever any relocatable element is added, as will happen with the TPLGENERATE call.) In any case it must precede any collection (i.e. a @MAP card) requiring the multiple package. The third card listed, which copies the description deck for the multiple package to tfile, is of course unnecessary if tfile and mfile coincide.

Description of the Package Elements

The heart of the triplex package is the collection of Fortran and assembler subroutines which do all the computations involving triplex variables. The source code for these routines is contained in the symbolic elements SOURCE/TPS and SOURCE/TPL for the single and multiple precision versions, respectively, and the resulting relocatable elements are included individually. The single precision package uses four additional routines whose assembler source and relocatable elements are included. These are BESTPACKAGE, borrowed intact from the interval arithmetic package at MRC to do the directed-rounding arithmetic, and INTCRI, BPACVO, and BPAHDE, all borrowed intact from the interval I/O package for the binary-decimal conversions. Since the multiple precision version routines all use multiple precision variables, they must all be translated by AUGMENT before they are to be compiled. The translated and compiled routines have been collected together to form the relocatable elements TPSPACKAGE and TPLPACKAGE, for the single and multiple precision versions, respectively. The MAP control cards for these collections are contained in the symbolic elements of the same name.

The absolute element TPLGENERATE is responsible for generation of a new triplex package as described above, and is the result of the collection of the two relocatable elements SDFIO and GENERATE, whose UNIVAC 1100 Assembler source elements are included under the same names. SDFIO is borrowed intact from the multiple package, while GENERATE is an adaptation of multiple's NEW-MACHINE. The symbolic element SKELETON/TRIPLEX is a skeleton runstream which is referenced by TPLGENERATE and passed to the EXEC for processing.

The symbolic elements DESCRIPTION/MULTIPLE, DESCRIPTION/TPS, and DESCRIPTION/TPL are the description decks for the multiple and triplex

packages as required by the AUGMENT precompiler. The multiple description deck must be maintained by the programmer in charge to reflect storage requirements for the multiple precision package being used unless the triplex package and the multiple package reside in the same file, in which case it will be updated as necessary by NEW-MACHINE.

The last element is the symbolic element for this document, called TPLDOC.

Listings

Listings of all programs in the triplex package are available on microfiche from the Mathematics Research Center.

Modifying the Triplex Package

If modifications or additions are to be made to the triplex package one need only write (or modify) the appropriate subroutines and make any necessary changes in the description deck for AUGMENT. Details on the format of the description deck and general guidelines for the subroutines can be found in [4]. The following remarks apply only to the triplex package:

The original routines have all been written to be "safe". That is, no parameter in the calling sequence is changed before all data is extracted from the parameters, so that expressions such as

```
A = A/A
```

do produce correct code. If any subroutines or functions are written or modified so as to be no longer safe, the description deck must reflect this.

Communication with the existing error package is through both the calling sequence and a common block. The call is always via:

```
CALL TPxERR (ICALLD,T1,T2,T3) ,
```

where x is S or L in the single and multiple precision versions

respectively, ICALLD is the name of the calling routine (e.g. 6HTPSADD), and T1, T2, T3 are three parameters, usually the three parameters which are placed in array PARAMS of common block TFAULT. The fault code itself is passed through common. In the single precision version it is stored in the integer variable IFAULT, the first word of common block INTFLT. In the multiple precision version it is stored in the integer variable MFAULT, the third word of common block MPIFLT. (INTFLT and MPIFLT are used for fault handling in the interval arithmetic and multiple precision packages.) Note that the error handlers treat each type of error separately, so as to accept different parameters and to produce different forms of output. This should make it relatively easy to include handling for newly defined errors. If the error handlers are modified, for example to incorporate additional error conditions, special care should be taken with regard to the nonstandard usage as mentioned above in *Error Conditions and Handling*.

The multiple precision arithmetic package has its own error handling routines which will produce error messages if they are not explicitly suppressed, as is done in the existing triplex subroutines. Most of these messages are controlled by the value of an interrupt mask, which should be set to zero to inhibit messages (the multiple fault flag is still set when errors are encountered). Subroutines MPIGMK and MPIMSK permit the user to get the old mask and to set the mask to a new value, respectively, in each case through the single parameter in the call. Messages arising in binary-decimal conversions (in either direction) are controlled by an internal logical variable - messages are printed if and only if its value is TRUE. This variable can be reset and its previous value obtained by a call to logical function MPIPRT. The value of the function is the previous value, and the new value is specified by the single parameter in the call. Details for both cases are in [3].

If any routines are to be added to or deleted from either package, appropriate changes must be made to the MAP control cards for that package. Note also that these control cards reference elements by element-name only - no file names are included. To get a proper collection, insert the following card before any collection:

```
@USE  TPF$,.tfile .
```

where "tfile" is the file containing the triplex package. It will probably be desirable to free the filename TPF$ using the card:

```
@FREE,A  TPF$ .
```

after the collection to avoid any undesired consequences of using TPF$ to refer to "tfile".

Disclaimer

This package has been tested and is believed to be correct and adequately documented. However, neither the programmer, the Mathematics Research Center, nor the University of Wisconsin assume any responsibility for any errors, omissions, malfunctions, or difficulties which may arise in its use.

REFERENCES

1. W. Binstock, J. Hawkes, and N.-T. Hsu, An interval input/output package for the UNIVAC 1108, The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report #1212, September 1973.
2. F. D. Crary, Multiple Precision Arithmetic Design with an Implementation on the UNIVAC 1108, The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report #1123, May 1971.
3. F. D. Crary, Multiple Precision Arithmetic Package-Revised FORTRAN Interface. Available from the author at the Mathematics Research Center, 610 Walnut St. (263-2520).
4. F. D. Crary, The AUGMENT Precompiler, I. User Information, The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report #1469, December 1974. (Revised April 1976)
5. F. D. Crary, The AUGMENT Precompiler, II. Technical Documentation, The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report #1470, October 1975.
6. Fortran V Library Functions, Reference Manual for the 1108, First Revision, The University of Wisconsin Computing Center, August 1971.
7. W. N. Kahan, A Survey of Error Analysis, "Proc. of IFIP Congress 1971, Vol. 11", Ed. by C. V. Freiman, J. E. Griffith and J. L. Rosenfeld, North-Holland Publ. Comp., Amsterdam, 1214-1239 (1972).
8. R. Krawczyk, Newton-algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken, Computing 4, 187-201 (1969).
9. T. D. Ladner and J. M. Yohe, An interval arithmetic package for the UNIVAC 1108, The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report #1055, May 1970.
10. Utility Routines, Reference Manual for the 1108, First Revision, The University of Wisconsin Computing Center, February 1971.

11. J. H. Wilkinson, Rounding errors in algebraic processes, London 1963.

12. J. M. Yohe, Rigorous bounds on computed approximations to square roots and cube roots, The University of Wisconsin-Madison, Mathematics Research Center, Technical Summary Report #1088, September 1970.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: MRC-TSR-1732	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER: Technical
4. TITLE (and Subtitle): A FORTRAN-TRIPLEX-PRE-COMPILER BASED ON THE AUGMENT PRE-COMPILER.	5. TYPE OF REPORT & PERIOD COVERED: Summary Report, - no specific reporting period
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): K. Boehmer and R. T. Jackson	8. CONTRACT OR GRANT NUMBER(s): DAAG29-75-C-0024
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Mathematics Research Center, University of Wisconsin, 610 Walnut Street, Madison, Wisconsin 53706	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 8 (Computer Science)
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office, P.O. Box 12211, Research Triangle Park, North Carolina 27709	12. REPORT DATE: Mar[redacted]77
	13. NUMBER OF PAGES: 30
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

interval analysis
interval arithmetic

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Triplex arithmetic is a variation of interval arithmetic in which a "main" or rounded value is computed in addition to the end-points of the containing interval. The "main" value may be considered, depending on the application, to be the "most probable" result of the computation, with the interval bounds indicating the possible error.

This paper describes an implementation of Triplex arithmetic in single and multiple precision. The implementation is based on the Augment precompiler to obtain an easily used package.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 224 200

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)